Formal Verification of Restoring Dividers made Fast and Simple

Jiteshri Dasari and Maciej Ciesielski
University of Massachusetts Amherst, MA, USA
jdasari@umass.edu, ciesiel@umass.edu

***Abstract*—The paper describes a formal verification method for hardware implementation of restoring divider circuits. The method is based on setting select signals to predefined constants to reduce the design to easily verifiable circuit components, followed by their verification using standard equivalence checking and SAT. It is then concluded by a global proof that the composition of those components indeed implements a divider. In contrast to previous approaches, the verification is done on a functional level without any reverse engineering of the internal structure. The results show significant improvement in verification time compared to other methods. The proposed approach can also be used in debugging by localizing the source of a bug. This feature is currently not available in the existing verification tools and will be a subject of future work.**

I. Introduction

Division plays a major role in several domains, including computer arithmetic, computational geometry, encryption, and other special purpose applications. Considerable progress has been made in recent years in verification of arithmetic circuits, such as multipliers, multiply-accumulate, and other components of arithmetic data-path, both in the integer and finite field domain [1][2]. However, with the exception of theorem provers and inductive non-automated systems that concentrate on proving the correctness of the underlying algorithms and on the resulting architecture [3], there has been little work devoted to dividers [4] [5] [6] [7].

This paper describes a formal verification method for gate-level implementation of restoring array divider circuits, the divider type commonly used in commercial applications. The method is based on setting select external signals to constants to reduce the design to easily verifiable circuit components, such as those involved in quotient bit and partial remainder generation. The verification of those component is combined with high-level reasoning about the correctness of the entire circuit composed of such components under the imposed input constraints. This approach is in line with the one given in a [8] advocating a "step-wise verification of circuit sub-components leading to polynomial complexity".

The main contribution of the paper is that it replaces the memory-intensive symbolic computer algebra (SCA) and algebraic rewriting techniques, successfully used in verification of integer and Galois Field multipliers [1][9][10][2], with a novel method specifically applied to dividers.

II. Background

The most advanced approach to arithmetic circuit verification is based on SCA and algebraic rewriting. Initially proposed in [11], it is based on representing the circuit in the algebraic domain, where both the circuit specification and the logic gates of its hardware implementation are represented as pseudo-Boolean polynomials. In this method, the polynomial representing the output bit vector is rewritten using polynomial representation of the logic gates all the way to the input bits, where it is compared to the circuit specification to reason about its correctness. The method has been successfully used by numerous authors in verification of multipliers [12] [1][10][13]. The method, however is plagued by excessive number of polynomials generated during the rewriting. Even though most of those polynomials eventually get reduced to zero (hence called *vanishing* polynomials), they can cause memory overload prohibiting the rewriting from completion. These issues have been addressed to some extent in the above-mentioned papers in the context of multiplier verification, but the verification of dividers requires more insight.

Recently, a promising approach has been proposed to use a combination of symbolic computer algebra and SAT in a method called *SAT-based Information Forwarding* (SBIF) [6]. The method uses SAT to gain information about vanishing monomials by propagating logic information in the direction opposite to the rewriting, i.e., from inputs to outputs. The method relies on the observation that the signs of the operands involved in computing intermediate remainders are always opposite, which allows one to eliminate a large number of intermediate monomials produced during rewriting. This, however, applies only to non-restoring dividers, while the restoring dividers do not share this feature. Furthermore, this technique does not help in proving an important constraint on the value of the final remainder R relative to the divisor D, namely that $R < D$. The authors of [6] resort to BDDs and SAT to solve this problem, claiming that the size of the resulting BDD is linear in the number of variables. This, however is the most expensive part of the method as it requires time-intensive variable ordering and construction of the BDD.

The work published in [7] recognizes that the failure of the straightforward application of algebraic rewriting comes not only from the excessive number of vanishing monomials. Additional problem comes from the fact that all commercial divider circuits are optimized based on the input constraint $X < 2^{n-1}D$. This constraint is imposed on the design to limit the bit-width of quotient Q and divisor D to n and to avoid overflow of Q [14]. As a result, the MSB cells of the divider have unused (redundant) outputs, making them

"unclean" for rewriting. The polynomials associated with those redundant outputs are analogous to don't cares (DC) in logic circuits: certain input values at local adders/subtractors cannot occur under the required input constraint and can be used as *satisfiability don't cares*. The authors of [7] use forward signal propagation to detect such don't cares (DC) and remove the affected monomials before they get propagated further to avoid potential memory overload. Instead of computing all don't cares, they propose a method to select a subset of the DCs that will minimize the number of polynomials at that level.

To accomplish this, they extract atomic blocks of the divider (half and full adders) and use standard Boolean logic techniques to compute satisfiability DCs at the inputs to these blocks. When the size of the polynomial increases by some predefined rate, they stop the procedure, backtrack to the previous step, apply the ILP optimization to reduce the number of vanishing monomials, and continue with polynomial rewriting. This approach proved quite efficient in restricting the size of the intermediate polynomials during rewriting. It can handle non-restoring 256-bit and 512-bit dividers in 16.5 min and 162 minutes, respectively. This method, however, uses reverse engineering to extract the single-bit adder/subtractor (HA/FA) blocks to gain access to the inputs on which to apply don't cares. As such, it makes certain assumptions about the internal structure of the divider, which may vary depending on the design style.

The work of [4] addresses the verification of array dividers by extracting a high-level arithmetic model from the low-level circuit implementation. The resulting arithmetic operations are compared with the abstract model of the divider using structural matching. The technique applies column-based XOR extraction, which relies on a regular structure of the adder/subtracter trees and the presence of the sum generation and carry propagation components. A lack of those components at the right places indicates a bug. The method requires the divider circuit to have a well defined architecture, and the adders to be represented with XOR gates, which is often not the case in a synthesized circuit. Their method performs structural analysis to see if the circuit's structure matches that of a divider but it does not explicitly verify its actual arithmetic function. In contrast, the method described here actually verifies whether the circuit performs the division operation, regardless of the internal structure of its components. In practice, there is a need for both approaches: a complete functional verification on a higher level combined with equivalence checking against known reference subcircuits.

Yet another attempt to divider verification applies a controversial technique of *hardware rewriting* [5][15]. It accomplishes verification by appending the circuit with a block that implements an inverse function, followed by logic resynthesis. If the circuit under verification is correct, the resulting logic becomes trivially redundant. In case of dividers, the method works well when applied to individual layers of the divider (each producing a quotient bit), but not for the entire circuit. Its disadvantage, therefore, is similar to that of [7] that needs access to the internal layered structure of the divider.

In contrast, our method does not require any reverse engineering and instead relies on functional analysis of the circuit. Access to partial remainders to verify the flow is achieved by setting signals (derived from the quotient bits) to constants 0/1 in order to reduce the design to easily verifiable circuit components, regardless of their internal structure. This is then followed by their verification using equivalence checking or SAT; and a global proof that the composition of those components does implement a divider. As shown in Section VI Experiments, this approach can verify 256 and 512-bit restoring dividers in a matter of 17 seconds and 2 minutes, respectively (50× and 76× speedup compared to [7]); and a 1023-bit divider in 9.4 minutes, something that none of the methods could achieve.

III. Restoring Divider

We start by reviewing a typical architecture of a restoring integer divider, shown in Figure 1. The idea is to gain understanding of the underlying division algorithm without a need to rely on the internal structure.

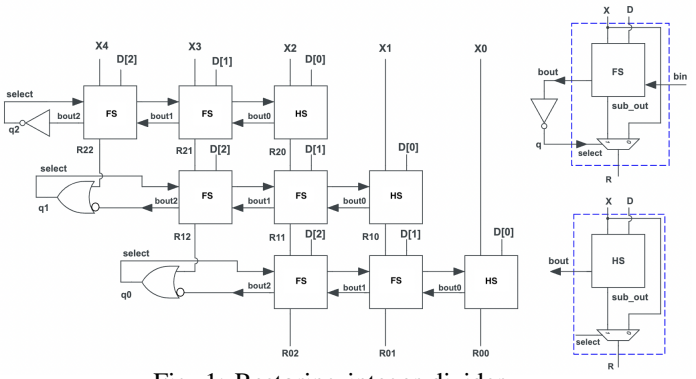

Fig. 1: Restoring integer divider.

The inputs are unsigned integer numbers. X is the dividend, D the divisor, Q the quotient and R the remainder. Typically, for an n-bit D, Q and R, the bit-width of a dividend is $2n$-1. To ensure the same bit-width of D and Q, and to guarantee that the resulting quotient Q will not overflow, a condition $X < 2^{n-1}D$ is imposed on the inputs, X, D [14]. With these constraints the optimized divider is implemented in an $n \times (2n-1)$ array structure shown in Figure 1. The division is performed by a repeated subtraction of the dividend by a divisor D shifted to the right by one bit after each subtraction. If the result of a given subtraction is non-negative, the subtraction takes place; otherwise the subtraction is not performed and the partial remainder is subjected to the next level subtraction.[1] The resulting partial remainder is then subjected to a controlled subtraction, with the last step producing the final remainder R.

Throughout the paper we use the following notation:

- Each layer i is labeled with the same index as the quotient bit q_i: top layer with n-1, bottom with 0.

[1]This is in contrast to a non-restoring divider, which performs subtraction and makes a correction in the next step by adding back the divisor D.

- Layer i has inputs R_{i+1}, D and outputs q_i, R_i.
- The input to the top layer is R_n, part of the dividend X.[2]
- The output of the bottom layer $R_0 = R$, the remainder.

The quotient bit q_i produced as the MSB of each subtractor layer serves as a *select* signal, which determines whether the input vector R_{i+1} or the difference $R_{i+1} - D$ is selected as the output, R_i. The verification process has two basic steps:

1) verify if the gate-level implementation computes a correct partial remainder R_i in each subtraction step; and
2) prove that the circuit composed of such controlled subtractions performs the required division under the required input constraint $X < 2^{n-1}D$.

Such a conceptually simple array structure turns out to be rather difficult for the SCA approach, mainly because of the dependence of quotient q_i on the n-bit subtractor with inputs R_{i+1} and D. Here is why.

The equation that characterizes layer i of the divider is:

$$R_i = R_{i+1} - q_i D \tag{1}$$

Backward rewriting at this level rewrites the output polynomial $R_i + q_i D$ into R_{i+1}. Note that the quotient bit q_i is a non-linear polynomial function of the n-bit subtractor inputs $R_i = \sum_{k=0}^{n-1} 2^k R_i(k)$ and $D = \sum_{k=0}^{n-1} 2^k D(k)$, which is then multiplied by D. This, in addition to the fact that some higher-significant output bits are not used (redundant), but should be included in the rewriting, causes an exponential increase in the number of monomials. The authors of [7] recognize this problem and attempt to solve it using satisfiability don't cares and ILP, but this is computationally expensive and, in our opinion, unnecessarily complex.

IV. Verification Approach

Instead of blindly rewriting the polynomials associated with layer i, we consider two separate cases, depending on the value of the the quotient q_i. We basically separate the output signal q_i from the control over the subtractor, and create a new input signal sel_i. This is done readily by manipulating the input Verilog or $blif$[3] file by replacing all the occurrences of signal q_i as input with a new input sel_i, while keeping q_i as the output quotient bit. When $sel_i = 0$, the input bits of R_{i+1} become connected directly to the output bits of R_i, i.e. $R_i = R_{i+1}$. On the other hand, when $sel_i = 1$, the layer performs subtraction, $R_i = R_{i+1} - D$. Verifying these two cases on the logic level is significantly simpler than performing rewriting. And, most importantly, access to individual bits of a given layer need not to be known or derived by structural extraction or reverse engineering, as in the earlier works [4][6] [7].

A. Case $sel_i = 0$: Vertical Flow Verification

This case can be implemented for each division step i (corresponding to a circuit layer i) by setting $sel_i = 0$ and resynthesizing the circuit. If in the resulting simplified circuit the input and output bits are directly connected, the layer correctly implements the condition $R_i = R_{i+1}$. This proves that the circuit for this layer correctly implements the design for the case when the subtractor generates $q_i = 0$.

It may first seem that in order to perform this check one needs to have access to the input and output signals, R_i and R_{i-1}, of the given layer. Fortunately, those signals can be readily discovered also by setting sel_i to 0. Only the input signals at the top layer (R_n part of the dividend X) and output signals at the bottom layer (the remainder, $R = R_0$) are needed. All the intermediate signals of partial products R_i can be learned from such generated circuit.

Specifically, starting with the top layer with signals R_n, setting $sel_{n-1} = 0$ should reduce the logic between R_n and R_{n-1} to bare wires. This automatically exposes the *signal names* of the next-level partial remainder, R_{n-1}, which are otherwise hidden and not recognizable in the original input (Verilog) file. The result of such a reduction can be seen in the circuit diagram of a 5-bit divider (called 5-3 divider, since $n = 3$) in Figure 2, marked with colored lines in the respective column bits. Logic synthesis tools, such as ABC [16], can accomplish this reduction, while also providing the needed signal names of the output vector R_{i-1}. This is shown in Figure 3 for this example. This process is then repeated for each layer, by selectively setting sel_i to 0 for that layer.

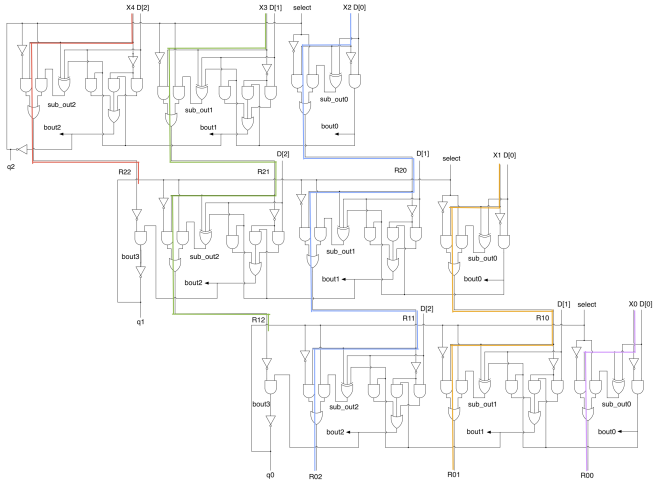

Fig. 2: Restoring divider - schematic

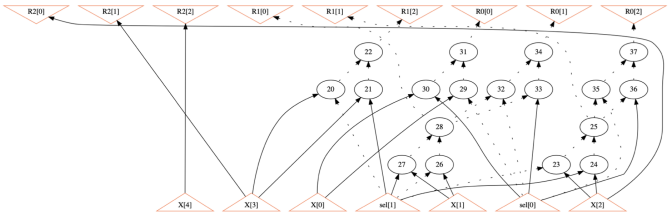

Fig. 3: Logic view after $sel_2 = 0$ reduction, top layer, showing direct connections: $X_4 \rightarrow R_2[2], X_3 \rightarrow R_2[1], X_2 \rightarrow R_2[0]$.

It should be noted that this important feature works only in a correct, *bug-free* circuit. If the propagation of constant $sel_i = 0$ does not simplify the logic by connecting the corresponding

[2]In our notation R is an n-bit vector, while the dividend X has $2n$-1 bits
[3]BLIF = Berkeley Logic Intermediate Format

inputs and outputs of that layer, there must be a bug on a logic path from R_{i-1} to R_i at this bit position. Simple analysis of the divider's logic indicates that this affects only the logic along the vertical data flow between the two layers (in this case a MUX), regardless of the internal structure of the divider. This is an important information that will help localize the bug, briefly discussed in Section VII.

In order for the signal names extracted in this manner to be trusted, we must first check if this logic path is correct. This can be checked by setting $sel_i = 0$ signal to all layers, $i = 1, ., n$. If this vertical path of the circuit is correct, then all the bits of Dividend X will be connected to some partial remainder bits at the respective layers; refer to the color-coded paths shown in Figure 2. Let us assume that this is the case, so we can use the extracted signal names of the partial remainders at each layer. Otherwise, more insight is needed to determine correct names of the boundary signals - a subject for future work on debugging.

B. Case $sel_i = 1$: Horizontal Flow/Subtractor Verification

We now need to verify the "horizontal" data flow that involves computing the difference $R_i = R_{i+1} - D$ across the entire row and generates the quotient bit q_i. This can be done by setting $sel_i = 1$, causing the layer i to act as a *subtractor*. At this point we already have access to the input R_{i+1} and output R_i signal names, along with the divisor D for each n-bit subtractor at level i. However, to perform such a verification, we also need access to the final *borrow-out* signal (b_{out}) of each subtractor. While this signal is readily available at the top layer, where $b_{out} = q'_{n-1}$, each subsequent layer has additional logic driven by b_{out} and the MSB of the partial remainder of the layer above it, namely $q_i = R_i[n-1] \vee b'_{out}$. For example, for the second layer of our 5-bit divider, $q_1 = R_2[2] \vee b'_{out}[2]$. To solve this problem, we set the MSB of R_{i+1} (from the layer above) to 0, which makes b_{out} visible as q'_i. In our example $b_{out}[2]$ can be obtained by setting $R_2[2] = 0$. The complement of signal $b_{out}[2]$ is then visible at q'_1.

At this point we have access to all the signals of the subtractor and can perform the verification using combinational equivalence checking (CEC) of ABC [16]. This potent tool can readily prove equivalence between our subtractor and a reference subtractor (generated by ABC or by any other means) for 256 bits and beyond in a matter of seconds.

C. Quotient Generation Verification

To complete the verification we also need to verify if the quotient bits Q are correctly generated by the circuit. Note that the MSB of Q, i.e., q_{n-1}, corresponding to the top layer, has been already indirectly verified, by imposing the input range constraint $X < 2^{n-1}D$, which imposes $q_{n-1} = 0$. Specifically, the constraint $X < 2^{n-1}D$, or equivalently $X - 2^{n-1}D < 0$ represents subtraction of the divisor D shifted to the left by $n-1$ bits. This is performed in the top layer and the result of this operation should be negative. That is, the b_{out} generated at this level should be 1, and the corresponding q_{n-1} should be 0. In principle, this condition can be verified by constructing a circuit $Z = \{X < 2^{n-1}D\}$ derived from the top layer, and checking if the Boolean expression $Z \wedge q'_{n-1}$ is constant 1; or, equivalently if $Z' \vee q_{n-1}$ is unSAT. However, this condition is automatically satisfied, as long at the top level subtractor is verified to be correct. in fact, this has been already done by "horizontal" verification of each layer (subtractor), described in Section IV-B.

We now need to verify if the remaining layers also generate a correct quotient q_i. This can be achieved simply by checking if each signal q_i computes the required logic, $q_i = R_{i+1}[n-1] \vee b'out_i$, expressed in terms of the respective bits R_{i+1} and $bout_i$. Note that the names of those signals are already available, derived in the previous steps. This is a trivial operation that requires checking equivalence of such a simple logic with the corresponding reference expression. We should emphasize that logic expression for q_i is known for a given type of the divisor (in this case restoring) and is independent of the particular implementation structure. For example, the logic for q_i shown in our 5-bit divider schematic in Figure 2 is $q_i = \overline{R' \wedge b_{out}}$, which is equivalent to $q_i = R \vee b'_{out}$. Hence, we are not extracting any logic gates but verifying if q_i satisfies the needed logic expression.

V. Global Flow

We conclude our verification with a proof that, once each layer has been verified to implement a controlled subtractor, the entire array indeed implements a divider, i.e. satisfies the characteristic equation:

$$X = Q \cdot D + R, \quad \text{and} \quad 0 \leq R < D \tag{2}$$

First we we need to revisit Equation (1) that characterizes layer i of the divider as an n-bit subtraction: $R_i = R_{i+1} - q_i D$. The divider performs n successive n-bit subtractions, each time shifting one bit to the right, so that the combined length of the input operand (dividend X) is $2n-1$ bit wide.

For this, we introduce a word-level version of the partial remainder, labeled PR_i, each $2n-1$ bit long, and rewrite Equation (1) as

$$PR_i = PR_{i+1} - q_i 2^{i-1} D \tag{3}$$

First, recall that in order to satisfy the input range constraint, $PR_n = X < 2^{n-1}D$, quotient bit q_{n-1} must be 0. With this, the output of the first layer $PR_{n-1} = PR_n$. The remaining $n-1$ layers implement a controlled subtraction $PR_{i-1} = PR_i - q_{i-1}2^{n-1}D$, which generate the remaining bits of quotient Q and the final remainder $PR_0 = R$. A simple proof is provided here for the 5-3 divider (with $n = 3$), in Figure 2. In this example the input constraint is $X < 4D$, which requires $q_2 = 0$.

The following arithmetic operations are implemented by the three layers of the divider, with the upper bound on each partial remainder provided on the right (to be discussed next):

$$\begin{aligned} PR_3 &= X && < 4D \\ PR_2 &= PR_3 - q_2(4D) && < 4D \end{aligned}$$

$$PR_1 = PR_2 - q_1(2D) < 2D$$
$$PR_0 = PR_1 - q_0(D) < D$$
$$R = PR_0$$

By adding both sides of the above expressions for PR_i (and ignoring for a moment the inequalities on the right) we obtain:

$$R = X - (4q_2 + 2q_1 + q_0)D$$

or equivalently $X = QD + R$. This proof can be easily generalized to dividers with arbitrary bit-size n. Basically, for each layer $i = n, ..., 1$, we have

$$PR_{i-1} = PR_i - q_{i-1}(2^{i-1}D) \quad (4)$$

Then, by induction, we obtain $PR_0 = PR_n - QD$ or, equivalently, $PR_n = QD + PR_0$, i.e., $X = QD + R$.

Finally, we need to take a look at the upper bounds on the partial remainders PR_i. It can be shown that it decreases at each iteration by half, finally leading to the condition $0 \leq R < D$. This is summarized by the following lemma:

Lemma: Given a restoring divider X/D with n layers and the input constraint $X < 2^{n-1}D$, the upper bound on the size of partial product PR_i for $i = n$-1, ..., 0 is determined by the following formula

$$PR_i < 2^i D$$

hence, for $i = 0$ $R < D$.

Proof by induction on i. In the following, refer to Equation 4.

Initial step: with $PR_{n-1} < 2^{n-1}D$, the formula is true for $i = n-1$. We now prove that this implies to be true for $i-1$, i.e., $PR_{i-1} < 2^{i-1}D$, where $PR_{i-1} = PR_i - q_{i-1}(2^{i-1})D$. Consider two cases:

- Case 1: $2^{i-1}D < PR_i < 2^i D$. In this case subtracting $2^{i-1}D$ from PR_i gives a positive result, bounded by $2^{i-1}D$, and $q_{i-1} = 1$ is generated.
- Case 2: $0 \leq PR_i < 2^{i-1}D$. In this case subtracting $2^{i-1}D$ would generate a negative result, so no subtraction is performed and $q_{i-1} = 0$.

In both cases, the lowest upper bound on PR_{i-1} at this level is $2^{i-1}D$, i.e. $PR_{i-1} < 2^{i-1}D$. Hence $PR_0 = R < D$. QED

VI. Experiments and Results

The verification method described in this paper was implemented as program written in Python with interfaces to ABC. The designs used for the experiments were generated in-house, written in Verilog HDL. They were first synthesized using Yosys [17], which provides a more robust parser and can handle larger subset of Verilog than ABC. The resulting files in $blif$ format were synthesized again by ABC to provide a cleaner version of $blif$ files. Python program then automates the required verification. The program was tested on the restoring divider circuits with dividends ranging from 5 to 1023 bit-widths.

The results shown in Table I give the CPU times in seconds for the individual steps needed to achieve the complete verification and the total CPU time. A "-" in the table indicates unavailable results. The results of our verification are compared in Column 2 with [15], which uses a technique called "hardware rewriting". This technique relies on the explicit knowledge of the layered structure of the divider and rewrites each row separately. The main idea is to add a circuit that reverses the computation of the partial remainder implemented by a single layer, called signature linearizer. This is then followed by resynthesizing the combined circuit, which should result in a trivial circuit with wires connecting the partial remainder inputs and outputs of the layer. Although this technique of layered hardware rewriting is efficient and scalable, its main drawback is that it requires the explicit layering structure. No results beyond 255 bits were available.

Columns 3-6 summarize the performance of our method. Column 3 gives the CPU times of synthesis when all the select bits $\{sel_i\}$ derived from quotient bits $\{q_i\}$ for the entire divider are set to 0, and Column 4 when each select bit is set selectively to extract a single layer. Column 5 gives the total CPU time to perform equivalence checking (CEC) when $sel_i = 1$ is set, one by one, for each subtractor. The reference circuits used in CEC are subtractors generated by ABC with the respective bit-width for divisor size n. Finally, Column 6 gives the final verification time for our method. All the times given in each column reflect the respective CPU time for all layers.

The last two columns show the results published in [7], with data for dividers up to 511-bit dividend (available only for non-restoring dividers). As one can see, our method can verify 256- and 512-bit restoring dividers in a matter of 17 seconds and 2 minutes, respectively, offering a 50× and 76× speedup, respectively, compared to [7]. Our method can also handle a 1023-bit divider in 9.4 minutes, something that none of the methods could complete. The data from [4] are not shown in the table, as they were available only for dividers with up to 65-bit dividends; and the CPU times for those dividers were several orders of magnitude larger.

VII. Extensions and Future Work

The proposed verification method, besides its high speed, low memory overhead and effectiveness, has additional important feature: it naturally supports debugging.

Assume that the divider has a logic error in one of the cells (say k) of the subtractor in one of the layers (l). This will cause the horizontal verification to fail during combinational equivalence checking (CEC) at that layer. Having already access to all the partial product signals of the layer, we can scan the layer from its LSB to its MSB, and examine the internal logic of the cell (a half- or full-subtractor) one by one. To access a given cell, we need to isolate it from other cells, i.e., gain access to the $borrow_{in}$ and the $borrow_{out}$ signals, while the inputs $R_{l+1}[k], D[k]$ and the outputs $R_l[k]$ are already available from the earlier verification steps. This can be done by setting $R_{l+1}[i]$ and $D[i]$ at the neighboring cells i to some controlling values. Similarly, one can verify the subtractor control logic (MUX) of the buggy cell by setting some signals to make the cell observable. We can use some of the known test generation techniques developed by the testing

TABLE I: Restoring Divider Verification times in CPU seconds

Dividend bits	Layered HR-SAT time (s) [15]	Verification times in seconds (**this Work**)				Don 't care optimization [7]	
		Complete divider re-synth all sel bits=0	Step wise re-synth selectively set sel=0	Equivalence checking sel=1 in each step	**Total time** (re-synth+CEC)	No. of bits	Time (s)
5	0.09	0.04	0.12	0.13	0.29	4	0.16
7	0.12	0.04	0.16	0.18	0.38	8	0.48
11	0.18	0.04	0.23	0.27	0.54	-	-
13	0.21	0.04	0.26	0.30	0.60	-	-
17	0.27	0.04	0.34	0.40	0.78	16	1.91
19	0.40	0.04	0.39	0.45	0.89	-	-
21	0.44	0.04	0.42	0.50	0.96	-	-
23	0.48	0.04	0.47	0.56	1.06	24	3.82
33	0.68	0.05	0.65	0.80	1.50	32	6.79
63	4.48	0.08	1.28	1.66	3.02	64	28.86
95	12.96	0.14	1.98	1.97	4.09	96	70.22
127	18.56	0.22	2.66	3.76	6.64	128	148.18
255	123.46	0.89	5.89	10.52	17.30	256	989.91
511	-	3.76	13.76	108.97	126.49	512	9,668.70
1023	-	17.20	35.01	512.28	564.49	1024	TO (>24 CPU hours)

community to facilitate this. These debugging ideas are still in the conceptual phase and need to be thoroughly researched in future work.

VIII. Summary and Conclusions

The paper describes a novel approach for verification of integer divider using a hybrid/hierarchical approach. The verification is accomplished in three phases:

1) *Extracting layer boundaries*: accomplished by setting select signals derived from quotient bits to identify the boundaries of the internal components (subtractors). Contrary to the earlier methods that extract components based on a fixed internal structure of the circuit, our method does it by functional analysis.
2) *Low-level verification of essential components, layers*: based on a standard, fast combinational equivalence checking (CEC) techniques, supported internally by SAT. It proves the correctness of the partial remainder and quotient bits generation components.
3) *Global proof that the entire assembly implements a divider*: done in a word-level space to demonstrate that the overall flow of data performs the division.

The proposed method is directly applicable to other divider types and architectures, such as non-restoring dividers, by selecting the set of signals specific to that architecture to access the internal circuit components. The verification approach described here addresses a general sentiment that the verification problem for complex arithmetic circuits cannot be solved with a single verification technique in polynomial space and time. w.r.t. to the size of the circuit. However, as discussed in a recent paper [8], "With additional knowledge about the boundaries of components, polynomial verification becomes possible through the step-wise verification of sub-components and the use of different formal proof engines." We believe that our paper does exactly that.

Acknowledgement

This work has been supported by a grant from the National Science Foundation, Award No. CCF-2006465.

References

[1] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model," *IEEE TCAD*, vol. 39, no. 6, pp. 1346–1357, 2019.
[2] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases," in *DAC'14*, 2014, pp. 1–6.
[3] D. M. Russinoff, *Formal Verification of floating-point hardware design: a mathematical approach*. Springer, 2018.
[4] M. H. Haghbayan and B. Alizadeh, "A dynamic specification to automatically debug and correct various divider circuits," *INTEGRATION, the VLSI journal*, vol. 53, pp. 100–114, 2016.
[5] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "Functional verification of hardware dividers using algebraic model," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2019, pp. 257–262.
[6] C. Scholl and A. Konrad, "Symbolic computer algebra and sat based information forwarding for fully automatic divider verification," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
[7] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1110–1115.
[8] R. Drechsler and A. Mahzoon, "Preserving design hierarchy information for polynomial formal verification," in *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–7.
[9] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining sat and computer algebra," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, 2019, pp. 28–36.
[10] A. Mahzoon, D. Große, and R. Drechsler, "REVSCA-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.
[11] M. Ciesielski, W. Brown, D. Liu, and A. Rossi, "Function Extraction from Arithmetic Bit-level Circuits," in *IEEE Annual Symposium on VLSI*, July 2014, pp. 356–361.
[12] A. Mahzoon, D. Große, and R. Drechsler, "Polycleaner: Clean your polynomials before backward rewriting to verify million-gate multipliers," in *Proc. International Conference on Computer-Aided Design, ICCAD*, 2018, pp. 129:1–129:8.
[13] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability," 2022.
[14] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, 2002.
[15] A. Yasin, "Formal verification of divider and square-root arithmetic circuits using computer algebra methods," *PhD dissertation; University of Massachusetts, Amherst*, 2020.
[16] A. Mishchenko *et al.*, "Berkeley logic synthesis and verification group, abc: A system for sequential synthesis and verification," *http://www.eecs.berkeley.edu/~alanmi/abc*, 2007.
[17] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.